Atelier B

Traducteurs de l'Atelier B

Manuel Utilisateur

version 4.6

ATELIER B
Traducteurs de l'Atelier B Manuel Utilisateur
version 4.6

Document établi par CLEARSY.

Ce document est la propriété de CLEARSY et ne doit pas être copié, reproduit, dupliqué totalement ou partiellement sans autorisation écrite.

Tous les noms des produits cités sont des marques déposées par leurs auteurs respectifs.

CLEARSY
Maintenance ATELIER B
Parc de la Duranne
320 avenue Archimède
Les Pléiades III - Bât. A
13857 Aix-en-Provence Cedex 3
France

Tél +33 (0)4 42 37 12 70
Fax 33 (0)4 42 37 12 71

mail : contact@atelierb.eu

Table des matières

1 Description du manuel **1**

1.1 Objectif 1

1.2 Connaissances pré-requises 1

1.3 Vue d'ensemble du manuel 2

1.4 Comment utiliser ce manuel 2

1.5 Conventions et syntaxe 2

1.6 Documents associés 3

2 Présentation du logiciel **5**

2.1 Mission 5

2.2 Environnement recommandé 5

2.3 Services offerts 6

2.3.1 Préambule : Motivation de la traduction en deux passes 6

2.3.2 Service de traduction automatique d'une implémentation B0 en langage cible 8

2.3.3 Service d'édition des liens 8

2.3.4 Machines de base 9

3 Principes d'exploitation **11**

3.1 Les modes de fonctionnement 11

3.1.1 Utilisation du Traducteur au moyen de l'IHM* 11

3.1.2 Utilisation du Traducteur au moyen du mode batch* 12

3.1.3 Utilisation du Traducteur en ligne 13

3.1.4 Compilation et exécution du code produit 14

3.2 Les entrées et sorties 15

3.2.1 Messages générées par le Traducteur 15

3.2.2 Cas d'une utilisation en ligne ou en mode batch* 15

3.2.3 Cas d'une utilisation au sein de l'IHM* de l'Atelier B 15

3.2.4 Les fichiers 15

3.3 Précautions d'emploi 16

3.3.1 Avertissement important relatif a la preuve 16
3.3.2 Avertissement important relatif aux valeurs des constantes MAXINT et MININT 16
3.3.3 Taille des lignes produites 16
3.3.4 Compatibilité du Traducteur avec l'Atelier B 17
3.3.5 Nommage des modules et des projets dans le cas des Traducteur Ada et HIA 17

4 Scénarios d'exploitation 19
4.1 Développement d'un projet B natif 19
4.1.1 Principe 19
4.1.2 Spécifications informelles de l'exemple 19
4.1.3 Architecture du projet et code B 20
4.1.4 Intégration des composants dans l'Atelier B 25
4.1.5 Traduction unitaire du code cible produit 25
4.1.6 Edition des liens du projet 25
4.1.7 Compilation du code cible et exécution de l'application 26
4.2 Développement d'un projet hétérogène B/Langage cible 26
4.3 Développement d'un projet hétérogène B/HIA 28

5 Liste complète des services 31
5.1 Traduction unitaire d'une implémentation 31
5.2 Edition des liens d'un projet 31
5.3 Traduction d'un projet en Ada avec traces 31
5.4 Restriction d'usage des traducteurs 34

6 Glossaire 35

A Les machines de base 37
A.1 Principe 37
A.1.1 Définition 37
A.1.2 Utilité 37
A.2 Description des machines de base livrées avec l'Atelier B 37
A.3 Méthode d'écriture d'une machine de base 38
A.3.1 Méthode d'écriture de la spécification B 38
A.4 Méthode d'écriture du code cible (spécification et corps) 38
A.4.1 Méthode d'écrire du code cible en Ada 39
A.4.2 Méthode d'écriture de code cible en HIA 39
A.4.3 Méthode d'écriture de code cible en C++ 40
A.4.4 Méthode d'écriture de code cible en C 40

A.4.5 Le fichier “B Link File” 41

B Spécificités du language B0 accepté par le traducteur HIA 43

B.1 Introduction 43

B.2 Traduction des tableaux 43

B.2.1 Principe 43

B.2.2 Exemple 43

B.3 Traduction des articles 44

B.3.1 Principe 44

B.3.2 Exemple 44

B.4 Paramètres formels 45

Table des figures

2.1 Fichiers produits pour l'interface de `mach` 8
2.2 Fichiers produits pour le corps de `mach` 8

3.1 Traduction d'une implementation en mode batch 12
3.2 Traduction d'une implementation depuis la ligne de commande 14

4.1 Spécification et implémentation du composant pile 21
4.2 Spécification et implémentation du composant affiche_pile 22
4.3 Spécification et implémentation du composant interface_pile 23
4.4 Spécification et implémentation du composant demo 24
4.5 Graphe de dépendance du projet 25
4.6 Compilation d'un code Ada généré par le Traducteur Ada 26
4.7 Exécution du programme PILE 27
4.8 Correspondance entre les instances physiques du projet et leur chemin d'accès en C, C++ ou Ada 28

5.1 Traduction d'une implementation depuis la ligne de commande 31
5.2 Options utilisables lors de la traduction unitaire d'un composant 32
5.3 Options utilisables lors de l'édition des liens d'un projet 33

A.1 Machines de base livrées avec l'Atelier B 38
A.2 Traduction des types B0 en Ada, HIA, C et C++ 39
A.3 Exemple de fichier "B Link File"' 42

Chapitre 1

Description du manuel

1.1 Objectif

Ce manuel utilisateur s'applique aux logiciels suivants :

– Traducteur Ada, à partir de la version 4.6[1]
– Traducteur HIA, à partir de la version 4.6[2]
– Traducteur C/C++, à partir de la version 4.2[3]

Par la suite, lorsque nous ferons référence au logiciel Traducteur, cela signifiera que l'on évoque indifféremment le traducteur Ada, HIA, C ou C++. Lorsque l'on souhaitera différencier ces logiciels, on écrira explicitement le Traducteur Ada, le Traducteur HIA, le Traducteur C ou le Traducteur C++. De la même manière, lorsque l'on fera référence au langage cible, cela signifiera que l'on évoque le langage Ada, HIA, C ou C++.

Le but du manuel utilisateur est de mettre les connaissances requises à la disposition des personnes amenées à faire fonctionner le Traducteur. Il poursuit un double objectif :

– permettre à ces personnes de se former de façon progressive.
– leur servir de référence pour identifier les comportements de ces logiciels.

Pour ce faire, on expose les connaissances pré-requises, la façon de consulter le manuel selon le besoin de l'utilisateur, les conventions d'écriture utilisées et les lectures utiles.

1.2 Connaissances pré-requises

La lecture de ce manuel suppose que l'utilisateur soit formé au langage B et au langage cible ainsi qu'à l'utilisation de l'Atelier B et du compilateur cible.

[1] Le traducteur Ada traduit l'ensemble des implémentation B0 en code Ada conforme aux normes Ada-83[ADA-83] et Ada-95[ADA-95].

[2] Le traducteur HIA produit du code "High Integrity Ada", syntaxiquement conforme à la norme SPARK décrite dans [SPARK]. En contrepartie de quelques restrictions sur le langage B0 en entrée, il génère un code plus simple, très proche du B0 et plus "sécuritaire".

[3] Ce traducteur traduit l'ensembe des implémentations en code ANSI-C[ANSI-C] ou ANSI-C++[ANSI-C++].

1.3 Vue d'ensemble du manuel

Le chapitre 2 présente les objectifs du Traducteur. Les environnements supportés et la philosophie générale de traduction y sont exposés.

Le chapitre 3 détaille les principes d'exploitation du Traducteur. L'utilisation au sein de l'Atelier B est décrite, tout comme l'utilisation en ligne depuis un shell* . Le chapitre se termine sur l'évocation des précautions d'emploi à respecter lors de l'utilisation du Traducteur

Le chapitre 4 illustre au moyen d'un exemple simple le cycle complet de développement d'un projet B traduit en langage cible. Le cas d'un projet B dont la traduction en langage cible est intégrée dans un projet plus vaste écrit nativement en langage cible est évoqué.

Le chapitre 5 résume les options d'utilisation du logiciel. Il détaille ensuite le mode d'utilisation du Traducteur qui permet de gérer les configurations des projets au moyen de l'outil SCCS* .

Enfin, le chapitre 6 explicite les termes techniques utilisés dans ce document.

L'annexe A donne la marche à suivre pour développer des machines de base.

1.4 Comment utiliser ce manuel

L'utilisateur débutant du Traducteur peut, dans une première lecture, se contenter d'étudier les chapitres 2, 3 et 4. La mise en oeuvre des exemples présentés dans ce dernier chapitre constitue une illustration complète de l'utilisation du logiciel et doit permettre une prise en main progressive et complète du Traducteur.

Lorsqu'il sera familiarisé avec le logiciel, l'utilisateur averti trouvera dans le chapitre 5 un résumé des options d'utilisation du Traducteur.

1.5 Conventions et syntaxe

– Les "objets informatiques" tels que les noms de fichiers, de fenêtre ou les choix d'options dans les menus sont écris au moyen d'une police non-proportionnelle, comme dans l'exemple ci-dessous :

  La machine `MM.mch`.

  Les échanges d'entrées/sorties entre l'utilisateur et le logiciel sont décrits au moyen de la même police. Pour différencier les entrées des sorties, les messages produits par le logiciel seront précédés du signe `>`, comme dans l'exemple ci-dessous :

```
ls
> AA.mch    AA_1.imp  SCCS
```

– Les mots dont la signification est expliquée dans le glossaire *(chapitre 6, page 35)* sont suivis d'une astérisque, comme dans l'exemple suivant : "L'utilisateur de l'IHM* "

– Les paragraphes décrivant les spécificités d'un ou plusieurs traducteurs seront précédés du signe

  ↺ , et écrits dans une police spécifique, comme ceci :

  ↺ Cette section ne concerne pas le traducteur ...

1.6 Documents associés

La bibliographie *(page 47)* donne une liste d'ouvrages qui permettent à l'utilisateur débutant de se former à l'utilisation de l'Atelier B et du langage cible, et qui servent de référence à l'utilisateur averti.

Chapitre 2

Présentation du logiciel

2.1 Mission

La mission du logiciel Traducteur est de réaliser une traduction automatique des implémentations B0 d'un projet en code source cible. Le code cible produit peut soit être compilé afin de réaliser un projet indépendant, soit être intégré à un développement en langage cible natif[1].

Le Traducteur est capable de traduire en langage cible l'ensemble du langage B0. Il n'y a aucune restriction, notamment en ce qui concerne le nommage des identificateurs : les éventuels identificateurs qui entreraient en conflit avec le langage cible sont renommés par le traducteur.

Remarque importante : le traducteur HIA travaille sur la base d'un language B0 qui possède quelques spécificités qui sont détaillées dans l'annexe B

Ainsi, tout composant qui est analysé avec succès par le vérificateur de B0 peut être traduit en langage cible.

Dans la suite de ce manuel, on appellera implémentation B0 toute implémentation de composant pour laquelle le vérificateur de B0 s'exécute avec succès.

2.2 Environnement recommandé

Le Traducteur est destiné à être exécuté sur les mêmes plate-formes que l'Atelier B.

Le Traducteur génère un code cible portable, conforme aux normes en vigueur.

Des options du logiciel permettent de paramétrer le code généré pour s'adapter au système et au compilateur cible[2].

Dans sa version 4.6, le code produit par ce logiciel a été testé avec un compilateur GNU dans les environnements suivants :

- Station de travail Sun sous Solaris 2.5.1.
- Station de travail Sun sous Solaris 2.6.
- Station de travail HP sous HP-UX 10.20
- Micro-ordinateur type PC sous Linux 2.2.

[1] Cette fonctionnalité permet d'intégrer à un projet écrit en langage cible la traduction des éléments sécuritaires réalisés en B.

[2] On peut ainsi réaliser une activité de "traduction croisée"

Remarque importante : L'utilisateur doit posséder un environnement de développement pour le langage cible complet, aucun outil de compilation ou d'interprétation du langage cible n'étant livré avec le Traducteur.

2.3 Services offerts

2.3.1 Préambule : Motivation de la traduction en deux passes

Cette section présente des aspects techniques qui peuvent être ignorés dans un premier temps par un utilisateur débutant *(qui peut alors se rendre directement au paragraphe 2.3.2)*.

Le Traducteur permet de traduire automatiquement toute implémentation B0 en langage cible. Une implémentation B0 n'est pas traduite en un code source en langage cible en un seul appel (ou une seule passe) du traducteur.

On utilise deux passes :

- la traduction unitaire : au cours de cette première passe, chaque implémentation B0 est traduite en un fichier "objet"
  , indépendament des autres implémentations. Ces fichiers "objets" produits par la première passe peuvent être réutilisés par plusieurs projets, permettant ainsi la création de bibliothèques B.
- l'édition de liens : au cours de cette seconde passe, le Traducteur produit notamment les fichiers en langage cible (se reporter au paragraphe 2.3.3). Ces fichiers en langage cible sont liés au projet en cours et ne peuvent pas être réutilisés dans un autre projet.

Les motivations de ce mécanisme de traduction en deux passes (production des fichiers objets puis édition de liens) sont liées aux objectifs suivants du Traducteur :

- Traduction complète de l'ensemble du langage B0.
- Traduction fidèle et performante.

Traduction complète de l'ensemble du langage B0

Un certain nombre de problèmes de conflits d'identificateurs peuvent survenir lors de la traduction de B0 en langage cible :

- Le langage B0 distingue les identificateurs `ident` et `IDENT`, ce qui n'est pas le cas du langage Ada.
- L'identificateur `void` en B0 entre en collision avec le mot clef réservé `void` en C++.
- Le langage B0 autorise l'écriture d'identificateurs comportant plus d'un caractère '_' à la suite *(comme par exemple* `ident__ificateur`*)*. Ces identificateurs ne sont pas des identificateurs Ada valides.
- Certains identificateurs B0 valides entrent en conflit avec le langage cible. Par exemple, `package` est un identificateur B0 valide.

Ainsi, il est clair qu'une phase de résolution des conflits d'identificateurs, globale à un projet, est nécessaire afin de réaliser une traduction automatique et systématique du langage B0 en langage cible.

Traduction fidèle et performante

↺ Cette section ne concerne pas traducteur HIA qui gère différemment la traduction des tableaux et des articles. Voir l'annexe B pour plus de détails

La traduction fidèle des tableaux B0 impose de déterminer par inférence les types des tableaux utilisés et de générer automatiquement les "types tableaux" associés. Cette gestion doit être performante car deux tableaux qui ont le même type inféré en B0 doivent être traduits par deux tableaux qui ont le même type généré en langage cible, afin par exemple de pouvoir copier et comparer ces tableaux.

Le même problème se pose lors de la traduction des articles B0 : il faut inférer une déclaration du type article associé.

Une phase de génération et de résolution des types tableaux et articles, globale au projet, est donc nécessaire à la traduction fidèle et performante des éléments de ce type du langage B0 *(qui ne type pas les tableaux et les articles)* vers le langage cible : C, C++ ou Ada *(qui impose une déclaration explicite des types tableaux et articles)*.

Résolution des collages implicites et du renommage

La résolution des collages implicites et des renommages dans un langage fortement typé comme Ada ou C++ ne peut se faire qu'au cours d'une phase d'édition des liens pendant laquelle le Traducteur a une vue globale de l'ensemble d'un projet.

Création des instances de composants

↺ Cette section ne s'applique qu'au traducteur HIA

L'éditeur est lien est responsable de la duplication *(copie physique des fichiers)* des fichiers pour chaque instance des composants du projet.

Ainsi si on utilise dans un projet deux instances `M1` et `i1.M1` d'un composant `M1`, alors l'éditeur de liens crée deux paquetages :

– Le paquetage `M1`, dans `m1.ads` et `m1.adb`
– Le paquetage `i1_M1`, dans `i1_m1.ads` et `i1_m1.adb`. Ce paquetage est obtenu par copie du paquetage `M1` et remplacement de toutes les occurences de `M1` par `i1_M1`.

Valuation des paramètres formels

↺ Cette section ne s'applique qu'au traducteur HIA

Le traducteur HIA déclare les paramètres formels dans les paquetages associés. Il doit donc définir dans chaque paquetage qui a des paramètres formels non seulement le nom et le type de ces paramètres, mais aussi leur valeur.

Or la valeur effective d'un paramètre formel n'est connue que lors de l'importation du module par un autre module du projet *(exemple : dans* `M1`*, la clause* `IMPORTS M2(10)` *: c'est dans* `M1` *que l'on connait la valeur* `10` *du paramètre formel* `param` *de* `M2`*. Mais c'est dans* `M2` *qu'il faut écrire* `param : constant INTEGER := 10`*)*.

L'éditeur de liens doit donc mettre en place les valeurs effectives des paramètres formels,

Langage cible	Après traduction	Après édition de liens
Ada	`mach.str`	`mach.ads`
HIA	`mach.str`	`mach.ads`
C	`mach.spe`	`mach.h`
C++	`mach.spe`	`mach.h`

FIG. 2.1 – Fichiers produits pour l'interface de `mach`

Langage cible	Après traduction	Après édition de liens
Ada	`mach.bod`	`mach.adb`
HIA	`mach.bod`	`mach.adb`
C	`mach.bdy`	`mach.c`
C++	`mach.bdy`	`mach.cpp`

FIG. 2.2 – Fichiers produits pour le corps de `mach`

correspodant à l'utilisation des composants dans le projet B à traduire.

2.3.2 Service de traduction automatique d'une implémentation B0 en langage cible

Ce service prend en entrée une implémentation B0 `mach.imp`, et produit les fichiers suivants :

- Deux fichiers pour l'interface de `mach`. Un après traduction et un après édition de liens. Les noms des fichiers sont données par le tableau 2.1
- Deux fichiers pour le corps de `mach`. Un après traduction et un après édition de liens. Les noms des fichiers sont donnés par le tableau 2.2
- Un fichier objet qui décrit les symboles importés et exportés par `mach`. Par défaut, de fichier est nommé `mach.blf`. Ce fichier permet entre autres de produire la liste des substitutions à appliquer sur les fichiers objets d'interface et de corps afin de produire les fichiers cible finaux.

2.3.3 Service d'édition des liens

Cas d'un projet autonome

Dans le cas d'un projet autonome, le service d'édition des liens prend en entrée le nom de la spécification d'une implémentation destinée à être le point d'entrée d'un projet ainsi que le nom d'un module de lancement à créer, et crée :

- L'ensemble des fichiers sources *(interfaces et corps)* des composants du projet.
  ↺ DANS LE CAS DU TRADUCTEUR HIA, LES FICHIERS SONT DUPLIQUÉS POUR CHAQUE INSTANCE DE COMPOSANT.
- Le code cible du module du lancement du projet.
- L'interface et le corps du module `sets` (en ADA et HIA) ou `SETS` (en C/C++) pour les types tableaux inférés, les constantes (concrètes), les ensembles abstraits et les prédéfinis (`succ`, `pred`, `MAXINT`, `MININT`, ...)

↻ (pour le traducteur HIA, ce fichier ne contient que les ensembles et fonctions prédéfinis).
– Le fichier `makefile` permettant de générer le projet.

Pour être utilisable comme point d'entrée d'un projet, une implémentation doit posséder une et une seule opération, qui ne possède aucun paramètre tant en entrée qu'en sortie. Par contre, le nom de cette opération est libre.

L'éditeur de liens parcourt récursivement les liens d'importation de cette machine et traduit ainsi en code cible "terminal" l'ensemble des fichiers objets utilisés par le projet[3]. Les fichiers objets sont recherchés dans le répertoire `lang` du projet en cours de traduction, puis dans les répertoires `lang` des bibliothèques utilisées, et ce dans l'ordre de leur déclaration.

Cas d'un projet hétérogène

Dans le cas d'un projet hétérogène, le service d'édition des liens prend en entrée le nom de la spécification d'une implémentation destinée à être le point d'entrée d'un projet ainsi que le nom d'un module de lancement à créer, et produit :

– L'ensemble des fichiers sources *(interfaces et corps)* des composants du projet.
– Le code cible du point d'accès au projet B. Ce module permet d'initialiser l'ensemble des composants B, puis d'accéder à l'ensemble des données et des opérations de ces machines.
– L'interface et le corps du module `sets` (en ADA) ou `SETS` (en C/C++) pour les types tableaux inférés, les constantes (concrètes), les ensembles abstraits et les prédéfinis (`succ`, `pred`, `MAXINT`, `MININT`, ...),
– Un squelette de fichier `makefile` permettant de générer le projet.

L'éditeur de liens parcourt récursivement les liens d'importation de cette machine et traduit ainsi en code cible "terminal" l'ensemble des fichiers objets utilisés par le projet. Les fichiers objets sont recherchés dans le répertoire `lang` du projet en cours de traduction, puis dans les répertoires `lang` des bibliothèques utilisées, et ce dans l'ordre de leur déclaration.

Remarque importante : les bibliothèques doivent être traduites avant le projet qui le utilise ! Si ce n'est pas le cas, l'édition des liens du projet échoue car les fichiers objets de la bibliothèque sont inexistants.

2.3.4 Machines de base

Le Traducteur est livré avec les spécifications B et les fichiers objets d'un ensemble de machines de base permettant de réaliser facilement des entrées/sorties formatées, des tableaux indexés par des ensembles énumérés ou des intervalles, ...

Ces machines sont décrites dans l'annexe A.

[3]Par contre, les fichiers objets correspondant à des machines non utilisées dans le projet ne sont pas traduits.

Chapitre 3

Principes d'exploitation

3.1 Les modes de fonctionnement

Le Traducteur peut fonctionner dans les trois modes suivants :

– Dans le cadre d'une session de travail avec l'IHM* de l'Atelier B.
– Dans le cadre du mode batch* de l'Atelier B.
– En ligne.

3.1.1 Utilisation du Traducteur au moyen de l'IHM*

Traduction d'une implémentation B0

Pour traduire une implémentation `implementation.imp` d'un projet `proj` au moyen de l'IHM* de l'Atelier B, il faut effectuer les opérations suivantes[1] :

– Sélectionner au moyen de la souris l'implémentation `implementation.imp`.
– Appuyer sur le bouton `Translator`. Dans la fenêtre qui apparaît, sélectionner le langage cible avec `Language` et choisir `Selected Only` pour `Components`. Puis appuyer sur le bouton `OK`. La traduction est alors lancée.

Remarque : comme nous l'avons vu au paragraphe 2.1, un composant doit avoir passé avec succès l'étape de vérification de B0 pour pouvoir être traduit.

↺ Cette étape n'est pas réalisée pour le traducteur HIA qui intègre nativement un module de vérification du langage B0 qui lui est propre

Ainsi, si cette étape n'a jamais été effectuée, ou si le composant a été modifié depuis la dernière vérification, l'outil de vérification de B0 est tout d'abord appelé sur l'implémentation à traduire.

Cet outil peut à son tour provoquer l'appel du Type Checker sur cette implémentation. Si besoin est, les machines vues sont également analysées par cet outil.

Le paragraphe 3.1.2 donne un exemple des messages produits par le Traducteur Ada au cours de la traduction.

[1] Les opérations de création et de gestion de projet ne sont pas décrites dans ce manuel, le lecteur trouvera leur description dans [ATB1].

Traducteur	Commande
Ada	`adatrans implementation_1`
HIA	`hiatrans implementation_1`
C	`ctrans implementation_1`
C++	`c++trans implementation_1`

FIG. 3.1 – Traduction d'une implementation en mode batch

On peut, de la même façon, réaliser la traduction de plusieurs implémentations en une seule opération en les sélectionnant toutes.

Édition des liens globale au projet

Pour réaliser une édition des liens globale à un projet, on sélectionne au moyen de la souris le fichier qui sert de point d'entrée du projet. On appuie sur le bouton `Translator` et on choisit le langage cible avec `Language` et `All` pour `Components`. Le nom du module de lancement (point d'accès) produit est le nom du projet B.

Le paragraphe 3.1.2 donne un exemple des messages produits par le Traducteur Ada au cours de la traduction.

3.1.2 Utilisation du Traducteur au moyen du mode batch*

Traduction d'une implémentation B0

Pour traduire une implémentation `implementation_1.imp` d'un projet `proj` au moyen du mode batch* de l'Atelier B, il faut taper la commande donnée par la table 3.1 [2] :

Le Traducteur [3] produit alors une sortie du type suivant[4] :

```
>> Translating into ADA the file implementation_1
> Entering B0->Ada mode ...
> Creating B Extended Tree
> Creating package specifications (/home/B/projet/lang/implementation.str)
> Creating package body (/home/B/projet/lang/implementation.bod)
> Creating B Link file (/home/B/projet/lang/implementation.blf)
> Free B Extended Tree
>
>
> Translation into ADA successful
```

Si l'on réitère la commande sans changer le fichier source B0, on obtient un message du type :

```
> Component implementation_1 is already translated
```

[2] Les opérations de création et de gestion de projet ne sont pas décrites dans ce manuel, le lecteur trouvera leur description dans [ATB1].

[3] L'exemple donné est produit par le Traducteur Ada, la sortie est similaire pour les autres traducteurs

[4] L'exemple donné est produit en mode verbeux. Par défaut, on obtient une sortie plus concise.

On peut alors forcer la traduction du composant en désactivant la fonction de dépendance (commande `ddm` ou `disable_dependence_mode`) et en relançant la traduction. La commande inverse pour réactiver la fonction de dépendance est `edm` ou `enable_dependence_mode`.

Édition des liens globale au projet

Pour réaliser une édition des liens globale à un projet, on détermine le point d'entrée du projet *(dans notre cas, c'est* `entree_1.imp`*)* et on tape la commande :

Langage cible	Commande
Ada	`ada_all entree_1`
HIA	`hia_all entree_1`
C	`ccompile entree_1`
C++	`c++all entree_1`

Le Traducteur [5] produit alors une sortie du type suivant[6] :

```
> Entering project mode
> Calling B linker
> Entering project mode
> Analysing module entree
> (entree) exports (constantes)
> Analysing machine constantes
> Creating makefile
> Creating ada source code for executable module (projet.bod)
> Analysing instance this
>       This instance does not have a SEES clause
> Analysing instance this.ref_constantes
>       This instance does not have a SEES clause
> Analysing imported variables of instance this
>       This module does not import any variable
> Analysing imported variables of instance this.ref_constantes
>       This module does not import any variable
> Creating template for package sets
> Installing project
> Creating temporary bed file /tmp/blka04747
> Executing "/home/ada/bed/bed -s /tmp/blka04747 -i /home/B/projet/lang/makefile.blf
> -o /home/B/projet/lang/makefile"
...
> Executing "rm /tmp/blka04747"
> Freeing allocated objects
>
> ADA translation successful
```

3.1.3 Utilisation du Traducteur en ligne

Le chapitre 5 présente les options d'utilisation offertes lors de l'utilisation du Traducteur. Ce sont elles qui doivent être utilisées si l'on veut effectuer des traductions en ligne.

Remarquons que cette possibilité doit être réservée à des utilisateurs expérimentés car :

[5] L'exemple donné est produit par le Traducteur Ada, la sortie est similaire pour les autres traducteurs
[6] L'exemple donné est produit par en mode verbeux. Par défaut, on obtient une sortie plus concise.

Traducteur	Logiciel
Ada	tbada
HIA	tbhia
C++	tbcpp
C	tbcpp -c

FIG. 3.2 – Traduction d'une implementation depuis la ligne de commande

– Lors de l'utilisation en ligne, il faut fournir de nombreux paramètres : répertoires "base de données projet" (`bdp`) et "traduction" (`lang`), chemin d'accès de l'éditeur de liens (outil `bed`), répertoire contenant les informations du projet `LIBRARY`, ...
– Si on utilise le Traducteur en ligne, il faut gérer à la main la cohérence des fichiers. Ainsi, si les fichiers sources B sont modifiés, il ne faut pas oublier de leur faire passer les étapes de vérification de B0, ...

Le tableau 3.2 donne le nom de chaque traducteur.

Dans les exemples suivants, nous considérerons un projet B `Mon_Projet`, que l'on veut traduire en C++. On doit alors connaître les informations suivantes :

identificateur	signification
`${Mon_Projet}/lang`	Chemin d'accès au répertoire de traduction
`${Mon_Projet}/spec`	Chemin d'accès aux sources B
`${AB}/press/lib`	Chemin d'accès au projet `LIBRARY` fourni avec l'Atelier B
`Entree`	Le point d'entrée du projet

1. Cas d'une traduction unitaire : Traduction de l'implantation `Component_1.imp` du projet `Mon_Projet`.
```
tbcpp -i Component_1.imp -P ${Mon_projet}/lang
      -I ${AB}/press/lib/spec -L ${AB}/press/lib/lang/cpp -w
```
2. Cas de la traduction du point d'entrée du projet avec édition des liens : Traduction du point d'entrée `Entree_1.imp` et édition des liens.
```
tbcpp -o Mon_Projet -e Entree -E bed -P ${Mon_Projet}/lang
      -I ${AB}/press/lib/spec -L ${AB}/press/lib/lang/cpp  -w
```

Cet exemple ne remplace pas la lecture du chapitre 5 qui donne la liste complète des services.

3.1.4 Compilation et exécution du code produit

– Exécution sur la machine qui héberge l'Atelier.
Il suffit, dans une fenêtre shell* , de se placer dans le répertoire `lang` du projet et de taper `make`.
– Exécution sur une machine cible.
Il faut transférer sur cette machine le contenu complet du répertoire `lang` du projet et taper `make`. Ce répertoire contient l'ensemble des fichiers nécessaires à la compilation *(y compris les fichiers provenant de la traduction des bibliothèques.)*.

Le fichier `makefile` définit également le but `clean` qui permet de supprimer tous les fichiers binaires produits par le compilateur. Pour utilise ce but, taper `make clean`.

3.2 Les entrées et sorties

3.2.1 Messages générées par le Traducteur

3.2.2 Cas d'une utilisation en ligne ou en mode batch*

Le Traducteur produit des messages décrivant son fonctionnement dans la sortie standard *(i.e.* `stdout`*)*. Il produit éventuellement des messages d'avertissement sur cette même sortie standard, et des messages d'erreur sur la sortie d'erreur *(i.e.* `stderr`*)*.

Ainsi, un script qui lance un scénario de traduction complète peut récupérer les résultats dans `stdout` et les erreurs dans `stderr`, comme dans l'exemple suivant [7] :

```
#!/bin/sh
# Script qui traduit toutes les implémentations du répertoire courant
rm -f /tmp/res
rm -f /tmp/err
for f in *.imp
do
    echo ‘‘Traduction de $f’’
    tbhia -i $f >> /tmp/res 2>> /tmp/err
done

echo ‘‘Résultat de la traduction :’’
cat /tmp/err | less
echo ‘‘Erreurs survenues lors de la traduction :’’
cat /tmp/err | less
```

3.2.3 Cas d'une utilisation au sein de l'IHM* de l'Atelier B

Les messages sont intégrés à l'IHM* de l'Atelier B.

3.2.4 Les fichiers

Le Traducteur lit et écrit les fichiers en utilisant l'API* standard du système d'exploitation. Les points à surveiller sont donc :

- Vérifier que l'utilisateur qui lance le logiciel a les droits suivants :
  - droit de lecture dans les répertoires de source et `bdp` du projet des bibliothèques utilisées, et pour tous les fichiers de ces répertoires.
  - droit d'écriture dans les répertoires `lang` et `bdp` du projet.
- Vérifier que le système de fichiers du projet n'est pas plein.

Si ces consignes ne sont pas respectées, le Traducteur produit un message d'erreur explicitant le problème dû au système. Ces messages sont fournis au logiciel par le système. Il peut donc être nécessaire de configurer le système ou le compte de l'utilisateur pour modifier les caractéristiques de ces messages *(la langue utilisée par le système pour fournir les messages d'erreurs peut parfois être paramétrée par des variables d'environnement ou par d'autres méthodes.)*.

[7] On utilise le traducteur HIA dans cet exemple

3.3 Précautions d'emploi

3.3.1 Avertissement important relatif a la preuve

Le code cible généré par le Traducteur n'est valide que si les composants qui sont traduits sont complètement prouvés.

Le Traducteur permet de traduire des projets dont les composants ne sont pas complètement prouvés afin d'offrir aux utilisateurs plus de souplesse dans leur développement. Cependant, la traduction d'un composant non encore prouvé provoque l'émission d'un message d'avertissement, et le code généré comporte dans son entête un commentaire d'avertissement.

En effet, le code généré peut ne pas être compilable *(cas d'une erreur de conception en B qui ne se détecte qu'à la preuve)*, soit lever des exceptions au cours de son exécution *(cas par exemple d'un accès à un index invalide d'un tableau)*.

3.3.2 Avertissement important relatif aux valeurs des constantes MAXINT et MININT

Le Traducteur Ada permet de redéfinir les valeurs de `MAXINIT` et de `MININT` afin de pouvoir traduire du code destiné à un système cible dont l'architecture diffère de celle du système hôte.

L'utilisation d'une valeur de MAXINT et/ou de MININT différente de celle employée par le prouveur provoque un résultat non garanti.

Par défaut, le Traducteur est compatible avec le prouveur. Il est recommandé de prendre conseil auprès de ClearSy si vous souhaitez modifier les valeurs de ces constantes.

3.3.3 Taille des lignes produites

Certains compilateurs cible n'admettent en entrée que des fichiers dont les lignes ne dépassent pas une certaine taille.

Ainsi, le Traducteur contrôle la longueur des lignes qu'il produit. Cette longueur est paramétrée par l'utilisateur au moyen des options `-l` et `-t`, et dont les valeurs par défaut respectives sont `80` et `4`.

Si l'utilisateur modifie ces valeurs, il doit s'assurer :

– Que la taille maximum de ligne qu'il fournit n'excède pas les capacités de son compilateur cible.
– Que la taille maximum de ligne qu'il fournit n'empêche pas la génération du code. En effet, les appels d'opération sont préfixés par les noms de machines qui les définissent. On voit que l'on peut aisément provoquer une génération de lignes insécables longues en donnant des noms "longs" aux machines et aux opérations. Il faut veiller alors à la compatibilité entre ce choix et le choix de la taille maximale des lignes produites. L'outil "Vérificateur aux limites" de l'Atelier B peut aider l'utilisateur à fixer ces limites.

3.3.4 Compatibilité du Traducteur avec l'Atelier B

Il faut toujours veiller à disposer d'une version du Traducteur compatible avec les outils de l'Atelier B.

3.3.5 Nommage des modules et des projets dans le cas des Traducteur Ada et HIA

↻ Cette restriction ne concerne que les traducteurs Ada et HIA.

Les seuls conflits de noms que l'éditeur de liens ne peut pas résoudre sont les conflits qui surviennent dans les noms de modules. En effet, l'éditeur de liens ne peut pas renommer les modules car le langage Ada impose que les fichiers qui représentent une unité de compilation *(i.e. une procédure ou un paquetage)* aient le même nom que cette unité[8].

Par exemple, si un nom de module ou de projet est en conflit avec un mot-clé Ada, l'éditeur de liens ne peut pas résoudre le conflit et la traduction en Ada échoue.

Les restrictions qui s'appliquent aux noms de modules et de projet sont les suivantes :

– Dans un projet, aucun module ne peut avoir le même nom que le projet.
– Le nom d'un projet ne doit pas être en conflit avec le langage cible.

On rappelle que le langage Ada ne différencie pas les majuscules des minuscules, donc les règles évoquées ci-dessus sont à appliquer sans tenir compte de l'utilisation éventuelles de majuscules. Ainsi, un projet `PROJET` ne peut pas contenir de module `projet`, et un projet ne peut pas s'appeler `ENTRY`.

[8]Ici on entend par nom le nom du fichier, sans son extension

Chapitre 4

Scénarios d'exploitation

4.1 Développement d'un projet B natif

4.1.1 Principe

Nous allons dans cette section développer un exemple de projet B natif, afin d'illustrer les aspects suivants du développement d'un tel projet :

- Influence de la traduction sur l'architecture du code B écrit.
- Traduction unitaire des modules B du projet.
- Edition des liens du projet.
- Compilation et exécution du code généré.

Nous donnerons le mode d'emploi détaillé du Traducteur au moyen de photos d'écrans de l'IHM* de l'Atelier B.

4.1.2 Spécifications informelles de l'exemple

On veut tester la gestion d'une pile d'entiers. Pour cela, on écrit plusieurs modules :

- Un module qui initalise une pile d'une taille donnée et qui permet d'empiler ou de dépiler un élément.
- Un module qui affiche une pile.

L'exemple devra créer deux piles de taille différentes et prouver par affichage de leur contenu que les piles sont bien initialisées et que les procédures d'empilement et de dépilement fonctionnent.

Remarque importante : Cet exemple, et les codes sources B présentés ci-dessous pour sa réalisation, n'a aucune prétention au niveau du langage B. Il ne se veut pas un exemple de style de conception B, mais tout simplement un exemple complet de création de projet, avec traduction en langage cible puis exécution du code produit. Ainsi, pour gérer des piles non bornées, nous aurions pu les implanter sur la machine de base `BASIC_ARRAY_VAR` à la place d'un tableau statique borné.

↺ Les exemples ne sont pas adaptés à une traduction HIA car les types tableaux ne sont pas déclarés explicitement *(voir l'annexe B pour plus de détails)*. Après chaque exemple, nous expliciterons les modifications à réaliser pour réaliser une traduction au moyen du traducteur HIA

4.1.3 Architecture du projet et code B

Le composant pile

Le composant `pile` modélise une pile d'entiers naturels. La taille de la pile est paramétrable *(c'est un paramètre de la machine)*, sous réserve de ne pas dépasser 10 éléments car pour simplifier notre exemple, la pile est implantée sur un tableau statique de 10 entiers. Le composant offre deux opérations `empiler` et `depiler` qui permettent de gérer la pile.

La figure 4.1 donne le code source B de la spécification et de l'implémentation de ce composant.

↺ Pour traduire en HIA : il faut définir un type tableau explicite `type_tableau = (1..10) --> NAT` et utiliser ce type pour typer `la_pile`

Le composant affiche_pile

Le composant `affiche_pile` permet d'afficher une "pile" au format défini par le composant `pile` c'est à dire une pile implantée sur un tableau de 10 éléments. Ainsi, l'opération `affiche` qui permet d'afficher une pile prend en paramètre non seulement le tableau qui représente la pile, mais également la taille courante de la pile, i.e. le nombre d'éléments du tableau qui font partie de la pile.

Pour des raisons de présentation, cette opération prend également en entrée un message à afficher avant d'afficher la pile.

La figure 4.2 donne le code source B de la spécification et de l'implémentation de ce composant.

↺ Pour traduire en HIA : il faut définir un type tableau explicite `type_tableau = (1..10) --> NAT` et utiliser ce type pour typer `pile`. Les autres composants *(par exemple* `pile`*)* doivent typer leurs piles avec `type_tableau` pour que la traduction fonctionne.

Le composant interface_pile

Le composant `interface_pile` offre un niveau d'abstraction supérieur aux deux composants précédents. Il est paramétré par la taille de la pile à créer.

Il crée une pile de cette taille, l'affiche pour vérifier qu'elle est vide, la remplit avec des entiers consécutifs puis l'affiche pour vérifier qu'elle est pleine. La pile est alors vidée de ses élements puis affichée pour vérifier qu'elle est vide.

La pile est créée par importation du composant `pile`. Elle est remplie puis vidée au moyen des opérations `empiler` et `depiler` de ce composant. L'importation du composant `affiche_pile` permet d'afficher la pile.

La figure 4.3 donne le code source B de la spécification et de l'implémentation de ce composant.

↺ Pour traduire en HIA : il faut typer les piles avec le type `type_tableau` de `affiche_pile`

```
MACHINE
   pile(nb_elements)

CONSTRAINTS
   nb_elements : NAT &
   nb_elements >= 1 &
   nb_elements <= 10

CONCRETE_VARIABLES
   la_pile, haut_de_pile

INVARIANT
   la_pile : (1..10)-->NAT &
   haut_de_pile : NAT &
   haut_de_pile >= 0 &
   haut_de_pile <= nb_elements

INITIALISATION
   la_pile : : (1..10) --> NAT ||
   haut_de_pile := 0

OPERATIONS

   empiler(val) =
   PRE
      haut_de_pile < nb_elements &
      val : NAT
   THEN
      la_pile(haut_de_pile) := val ||
      haut_de_pile := haut_de_pile + 1
   END ;

   depiler =
   PRE
      haut_de_pile >= 1
   THEN
      haut_de_pile := haut_de_pile - 1
   END

END
```

```
IMPLEMENTATION
   pile_1(nb_elements)

REFINES
   pile

INITIALISATION
   la_pile := (1..10)*{0} ;
   haut_de_pile := 0

OPERATIONS

   empiler(val) =
   BEGIN
      haut_de_pile := haut_de_pile + 1 ;
      la_pile(haut_de_pile) := val
   END ;

   depiler =
   BEGIN
      haut_de_pile := haut_de_pile - 1
   END

END
```

FIG. 4.1 – Spécification et implémentation du composant pile

```
MACHINE
   affiche_pile(nb_elements)

CONSTRAINTS
   nb_elements : NAT &
   nb_elements >= 1

OPERATIONS
   affiche(
      message,
      pile,
      taille_de_pile) =
   PRE
      message : STRING &
      pile : (1..10) --> NAT &
      taille_de_pile : NAT &
      taille_de_pile <= nb_elements
   THEN
      skip
   END
END
```

```
IMPLEMENTATION
   affiche_pile_1(nb_elements)
REFINES
   affiche_pile
SEES
   BASIC_IO
OPERATIONS
   affiche(message, pile, taille_de_pile) =
   BEGIN
      STRING_WRITE("Resultat attendu : ") ;
      STRING_WRITE(message) ;
      STRING_WRITE("\nResultat effectif :\n") ;
      IF (taille_de_pile = 0)
      THEN
         STRING_WRITE("-- la pile est vide --\n")
      ELSE
         VAR
            ii
         IN
            STRING_WRITE("(fond de la pile) ") ;
            ii := 1 ;
            WHILE ii <= taille_de_pile
            DO
               INT_WRITE(pile(ii)) ;
               STRING_WRITE(" ") ;
               ii := ii + 1
            INVARIANT
               ii : NAT &
               ii >= 1 &
               ii <= (nb_elements + 1)
            VARIANT
               nb_elements + 1 - ii
            END ;
            STRING_WRITE("(haut de la pile)\n")
         END
      END
   END
END
```

FIG. 4.2 – Spécification et implémentation du composant affiche_pile

```
MACHINE
   interface_pile(nb_elements)

CONSTRAINTS
   nb_elements : NAT &
   nb_elements >= 1

OPERATIONS

   demonstration = skip

END
```

```
IMPLEMENTATION
   interface_pile_1(nb_elements)
REFINES
   interface_pile
IMPORTS
   pile(nb_elements),
   affiche_pile(nb_elements)
SEES
   BASIC_IO
OPERATIONS
   demonstration =
   BEGIN
      /*? On affiche la pile vide ?*/
      STRING_WRITE("Affichage de la pile vide :") ;
      affiche("PILE VIDE", la_pile, haut_de_pile) ;
      /*? On remplit la pile ?*/
      STRING_WRITE("On remplit la pile puis
         on l'affiche :\n") ;
      VAR
         ii
      IN
         ii := 1 ;
         WHILE (ii <= nb_elements)
         DO
            empiler(ii) ;
            ii := ii + 1
         INVARIANT
            ii : NAT & ii >= 1 &
               ii <= (nb_elements + 1)
         VARIANT
            nb_elements + 1 - ii
         END
      END ;
      /*? On affiche la pile pleine ?*/
      affiche("PILE PLEINE", la_pile, haut_de_pile) ;
      /*? On vide la pile ?*/
      STRING_WRITE("On vide la pile puis
         on l'affiche :\n") ;
      VAR
         ii
      IN
         ii := 1 ;
         WHILE (ii <= nb_elements)
         DO
            depiler ;
            ii := ii + 1
         INVARIANT
            ii : NAT & ii >= 1 &
               ii <= (nb_elements + 1)
         VARIANT
            nb_elements + 1 - ii
         END
      END ;
      /*? On affiche la pile vide ?*/
      affiche("PILE VIDE", la_pile, haut_de_pile)
   END
END
```

FIG. 4.3 – Spécification et implémentation du composant interface_pile

```
MACHINE
   demo

OPERATIONS

   demo_piles =
      skip

END
```

```
IMPLEMENTATION
   demo_1

REFINES
   demo

IMPORTS
   BASIC_IO,
   interface_A.interface_pile(3),
   interface_B.interface_pile(4)

OPERATIONS

   demo_piles =
   BEGIN
      STRING_WRITE("-- DEBUT DU TEST\n\n") ;

      STRING_WRITE("
         ------------------------------------------------\n") ;
      STRING_WRITE("
         Demonstration avec une pile de taille maximum 3\n") ;
      STRING_WRITE("
         ------------------------------------------------\n") ;
      interface_A.demonstration ;

      STRING_WRITE("
         \n\n------------------------------------------------\n") ;
      STRING_WRITE("
         Demonstration avec une pile de taille maximum 4\n") ;
      STRING_WRITE("
         ------------------------------------------------\n") ;
      interface_B.demonstration ;

      STRING_WRITE("-- FIN DU TEST\n\n")
   END

END
```

FIG. 4.4 – Spécification et implémentation du composant demo

Le composant demo

Le composant `demo` est chargée de lancer l'exécution de l'application. Il ne possède qu'une seule opération, qui créee puis teste les deux piles.

Ce composant sera le **point d'entrée** du projet B. C'est lui qui est la racine de l'arbre d'importation du projet, comme le montre la figure 4.5.

La figure 4.4 donne le code source B de la spécification et de l'implémentation de ce composant.

Entre autres, c'est lui qui importe la machine de base `BASIC_IO`[1] car plusieurs composants du projet vont produire des affichages à l'écran *(y compris le composant* `demo`*)* et nous savons qu'il faut respecter la règle "un ancêtre dans le graphe d'importation ne peut pas faire de `SEES`".

[1] Machine qui permet de réaliser des entrées/sorties, décrite dans l'annexe A

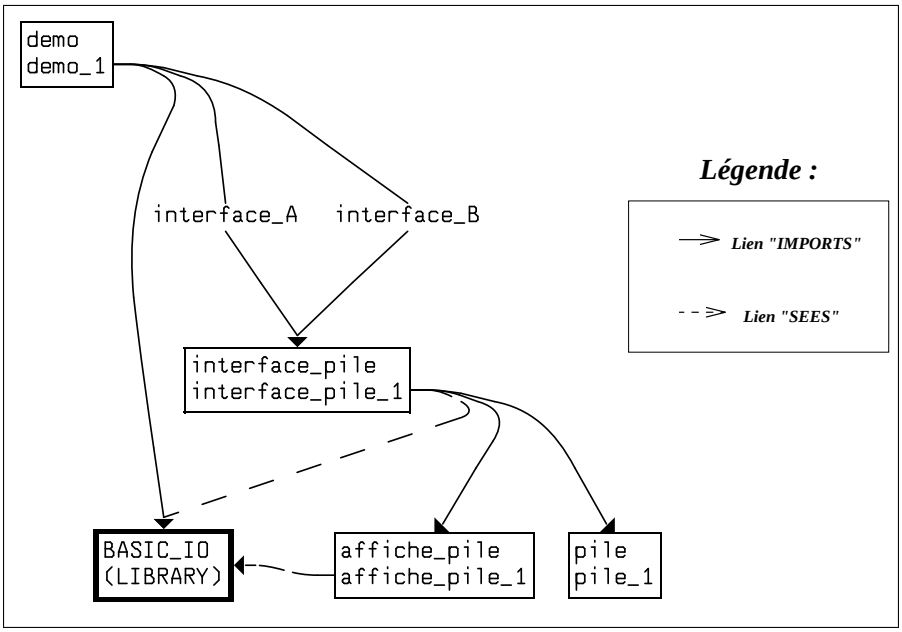

FIG. 4.5 – Graphe de dépendance du projet

4.1.4 Intégration des composants dans l'Atelier B

Le lecteur se réfèrera au document [ATB1] pour avoir des explications détaillées sur l'intégration de composants dans l'Atelier B. Ce paragraphe se borne à lister succintement les étapes à suivre.

Ainsi, l'utilisateur doit effectuer les opérations suivantes :

- Créer un projet. Dans la suite de ce chapitre, nous supposerons que le nom du projet est `DEMO_PILES`. Attacher à ce projet la bibliothèque `LIBRARY`. Le projet aura ainsi accès à la machine de base `BASIC_IO`.
- Insérer les composants détaillés au paragraphe 4.1.3. Effectuer les étapes de `Type_Check` puis de `B0 Check` sur ces composants.
  ↺ POUR LE TRADUCTEUR HIA, L'ÉTAPE DE B0 CHECK NE DOIT PAS ÊTRE RÉALISÉE.
- **Réaliser la preuve de ces composants**. Si cette étape n'est pas obligatoire pour pouvoir réaliser la traduction en lanage cible, elle est la seule garante de qualité du code produit. Le résultat de la traduction d'un code non prouvé à 100% n'est pas garanti.

4.1.5 Traduction unitaire du code cible produit

On sélectionne les quatre implémentations du projet et on appuit sur `Translator`. Dans la fenêtre qui apparaît, on choisit le langage cible et `Selected Only`.

4.1.6 Edition des liens du projet

On sélectionne l'implémentation `demo_1` et on appuit sur `Translator`. Dans la fenêtre qui apparaît, on choisit le langage cible et `All`.

L'éditeur de liens crée un module chargé de lancer le projet *(c'est le module de démarrage)*.

```
/home/ATELIER/B/pile/lang% ls
DEMO_PILES.adb        basic_io.ads        interface_pile.blf        pile.bod
DEMO_PILES.bod        demo.adb            interface_pile.bod        pile.str
affiche_pile.adb      demo.ads            interface_pile.str        sets.adb
affiche_pile.ads      demo.blf            makefile                  sets.ads
affiche_pile.blf      demo.bod            makefile.blf              sets.bod
affiche_pile.bod      demo.str            pile.adb                  sets.str
affiche_pile.str      interface_pile.adb  pile.ads
basic_io.adb          interface_pile.ads  pile.blf
/home/ATELIER/B/pile/lang% make
gnatgcc -I/home/ATELIER/B/LIBRARY/lang -c DEMO_PILES.adb
gnatgcc -I/home/ATELIER/B/LIBRARY/lang -c sets.adb
gnatgcc -I/home/ATELIER/B/LIBRARY/lang -c demo.adb
gnatgcc -I/home/ATELIER/B/LIBRARY/lang -c basic_io.adb
gnatgcc -I/home/ATELIER/B/LIBRARY/lang -c interface_pile.adb
gnatgcc -I/home/ATELIER/B/LIBRARY/lang -c pile.adb
gnatgcc -I/home/ATELIER/B/LIBRARY/lang -c affiche_pile.adb
gnatbl DEMO_PILES.ali -o DEMO_PILES
/home/ATELIER/B/pile/lang% ls -l DEMO_PILES
-rwxr-xr-x 1 ATELIER bin 205899 May 17 11 :35 DEMO_PILES*
```

FIG. 4.6 – Compilation d'un code Ada généré par le Traducteur Ada

Lors d'une utilisateur au sein de l'IHM* de l'Atelier B, ce module a le même nom que le projet.

4.1.7 Compilation du code cible et exécution de l'application

On ouvre une session sur la machine cible. Dans notre exemple, le système cible est identique au système de développement *(ce sont tous deux des systèmes Unix)*, le langage cible choisit est l'Ada et on utilise le compilateur Ada `gnat`.

On se place dans le répertoire où les fichiers sont traduits, et on tape `make`. La figure 4.6 donne un exemple d'affichage obtenu.

Avertissement : La traduction de ce projet avec le traducteur fourni avec la version 3.6 de l'Atelier B conduit à une erreur du traducteur.

On peut alors lancer l'exécution du projet. La figure 4.7 présente le résultat de cette exécution.

4.2 Développement d'un projet hétérogène B/Langage cible

↺ CETTE SECTION NE S'APPLIQUE PAS AU TRADUCTEUR HIA, QUI COMPORTE DES SPÉCIFICITÉS DÉCRITES AU PARAGRAPHE 4.3

Tout comme un projet autonome, un projet hétérogène doit comporter un point d'entrée. Le point d'entrée est un composant logiciel (appelé paquetage en Ada ou module en C,C++) qui permet de créer les instances physiques des composants et de les initialiser correctement. Il permet ensuite d'accéder à toutes ces instances. Il exporte une procédure `INITIALISATION` et selon le langage cible :

```
-- DEBUT DU TEST

-----------------------------------------------
Demonstration avec une pile de taille maximum 3
-----------------------------------------------
Affichage de la pile vide :Resultat attendu : PILE VIDE
Resultat effectif :
-- la pile est vide --
On remplit la pile puis on l'affiche :
Resultat attendu : PILE PLEINE
Resultat effectif :
(fond de la pile) 1 2 3 (haut de la pile)
On vide la pile puis on l'affiche :
Resultat attendu : PILE VIDE
Resultat effectif :
-- la pile est vide --


-----------------------------------------------
Demonstration avec une pile de taille maximum 4
-----------------------------------------------
Affichage de la pile vide :Resultat attendu : PILE VIDE
Resultat effectif :
-- la pile est vide --
On remplit la pile puis on l'affiche :
Resultat attendu : PILE PLEINE
Resultat effectif :
(fond de la pile) 1 2 3 4 (haut de la pile)
On vide la pile puis on l'affiche :
Resultat attendu : PILE VIDE
Resultat effectif :
-- la pile est vide --
-- FIN DU TEST
```

FIG. 4.7 – Exécution du programme PILE

Instance	Chemin d'accès en Ada	Chemin d'accès en C, C++
`demo`	`this`	`entry`
`BASIC_IO`	`this.ref_BASIC_IO`	`entry→ref_BASIC_IO`
`interface_A.interface_pile`	`this.ref_interface_A`	`entry→ref_interface_A`
`interface_B.interface_pile`	`this.ref_interface_B`	`entry→ref_interface_B`
`pile` créée par `interface_A.interface_pile`	`this.ref_interface_A.ref_pile`	`entry→ref_interface_A →ref_pile`
`pile` créée par `interface_B.interface_pile`	`this.ref_interface_B.ref_pile`	`entry→ref_interface_B →ref_pile`
`affiche_pile` créée par `interface_A.interface_pile`	`this.ref_interface_A.ref_pile`	`entry→ref_interface_A →ref_affiche_pile`
`affiche_pile` créée par `interface_B.interface_pile`	`this.ref_interface_B.ref_pile`	`entry→ref_interface_B →ref_affiche_pile`

FIG. 4.8 – Correspondance entre les instances physiques du projet et leur chemin d'accès en C, C++ ou Ada

– en Ada, un objet `this`,
– en C++, un objet `entry`,
– en C, une structure `entry`,

La procédure `INITIALISATION` permet de créer les instances physiques, et `this/entry` permet de référencer les objets du projet. Ainsi, `this/entry` référence l'instance du point d'entrée du projet *(i.e. l'instance de* `demo` *dans notre exemple)*, et ensuite on peut parcourir récursivement le graphe de dépendance du projet en appliquant la règle suivante :

– Si on atteint l'instance I du graphe grâce au chemin Λ.
– Si I utilise[2] une instance J
– Alors on atteint l'instance J par le chemin Λ`.ref_`J en Ada ou $\Lambda \rightarrow$`ref_`J en C, C++.

Ainsi, en se référant au graphe de dépendance illustré par la figure 4.5 de la page 25, on peut construire une table qui associe à chaque instance physique du projet un "chemin d'accès". La figure 4.8 présente cette table pour notre exemple.

On peut alors appeler les opérations des composants, sans oublier de leur passer en premier argument le paramètre d'instance implicite, i.e. le pointeur qui permet de les atteindre. Ainsi, on peut appeler la méthode `affiche` du composant `affiche_pile` crée par `interface_A` en écrivant le code suivant :

Langage	Code
Ada	`affiche_pile.affiche(this.ref_interface_A.ref_affiche_pile, ...) ;`
C	`affiche_pile→affiche(entry→ref_interface_A→ref_affiche_pile, ...) ;`
C++	`entry→interface_A→affiche_pile→affiche( ...)`

4.3 Développement d'un projet hétérogène B/HIA

↻ CETTE SECTION NE S'APPLIQUE Q'AU TRADUCTEUR HIA. LE DÉVELOPPEMENT DES PROJETS HÉTÉROGÈNES POUR LES AUTRES TRADUCTEURS EST DÉCRIT AU 4.2

Le code généré par le traducteur HIA est très simple et très proche du B0. Un paquetage est crée pour chaque instance de composant. Chaque paquetage a le même nom que le composant associé, éventuellement préfixé par le préfixe de renommage du composant, suivi par un caractère '_'.

[2] Au sens IMPORTS, SEES ou EXTENDS

Les constantes, les paramètres formels et les variables du composants sont traduites dans le paquetage. Les opérations ont le même nom et la même signature qu'en B.

Ansi le développement d'un projet hétérogène B/HIA n'impose aucun travail supplémentaire par rapport à un développement Ada "HIA" natif.

Chapitre 5

Liste complète des services

5.1 Traduction unitaire d'une implémentation

trad `[OPTIONS] -i nom_implementation[.suffixe]`, où `OPTIONS` est une combinaison des options présentées dans le tableau 5.2 et *trad* est donné par le tableau 5.1.

Traducteur	**Logiciel**
Ada	tbada
HIA	tbhia
C++	tbcpp
C	tbcpp -c

Fig. 5.1 – Traduction d'une implementation depuis la ligne de commande

5.2 Edition des liens d'un projet

trad `[OPTIONS] -o nom_executable -e spec_point_entree -E chemin_bed`, où `OPTIONS` est une combinaison des options présentées dans le tableau 5.3 et *trad* est donné par le tableau 5.1.

Remarque : Une option permet de changer le nom du compilateur cible lors de l'édition des liens. Le fichier `makefile` produit instancie la variable `ADA_COMPILE` *(ou* `CPP_COMPILE`*)* avec cette valeur. On peut toujours changer cette valeur après coup, soit en la modifiant directement dans le makefile, soit en la passant dans la ligne de commande, comme dans l'exemple ci-dessous :

```
make ADA_COMPILE=mon_compilateur_ada
make CPP_COMPILE=mon_compilateur_c++
```

5.3 Traduction d'un projet en Ada avec traces

↺ Cette section ne concerne que le traducteur Ada

Option	Sémantique	Valeur par défaut Traducteurs Ada/HIA	Valeur par défaut Traducteur C, C++
B	Change le suffixe des fichiers objets de type "corps de paquetage"	`bod`	`bdy`
D	Affiche la configuration avant de traduire		
C	Informations de compilation sur le logiciel		
I	Ajout un chemin d'accès pour la recherche des sources B		
l	Change la taille maximale en caractères des lignes produites	`80`	`80`
O	Change le suffixe des fichiers objets de type "B Link File"	`blf`	`blf`
P	Change le répertoire de sortie	`../lang`	`../lang`
S	Change le suffixe des fichiers objets de type spécification ou interface	`str`	`spe`
t	Change la valeur en caractères d'une tabulation	`4`	`4`
T	Ada : Génére du code dont l'exécution produit des traces sur les opérations appelées et les valeurs des paramètres		
v	Mode verbeux		
V	Numéro de version et usage du logiciel		
w	Ajoute le message "composant non prouvé" au code généré *(utilisé automatiquement par l'Atelier).*		

FIG. 5.2 – Options utilisables lors de la traduction unitaire d'un composant

L'option `-T` du Traducteur Ada permet d'effectuer une traduction du projet avec des traces. Cette option permet :

– Au niveau d'une traduction unitaire, de demander une traduction d'un module avec production de traces.
– Au niveau de l'édition des liens, de demander l'ajout des modules de trace lors de la compilation.

Un projet qui a été traduit avec des traces produit lors de son exécution une trace complète des appels de fonction. Cette trace est produite dans le fichier `.trace`, situé dans le répertoire d'où le projet est lancé. Dans ce fichier, on répertorie :

– Les appels de fonction, avec affichage de l'instance implicite ainsi que de la valeur des paramètres en entrée.
– Les retours de fonction, avec affichage de l'instance implicite ainsi que de la valeur des paramètres en sortie.

Ainsi, une traduction avec des traces permet à l'utilisateur de suivre précisément le déroulement de son projet sans avoir à rajouter pour cela des instructions dans son code source B.

Sémantique	Valeur par défaut Traducteurs Ada/HIA	Valeur par défaut Traducteurs C/C++
Change le nom du compilateur cible	A, `gnatgcc`	T, `gcc`
Change le suffixe des fichiers objets de type “corps de paquetage”	B, `bod`	B, `bdy`
Affiche les conflits de noms et leur résolution	c	c
Informations de compilation sur le logiciel	C	I
Affiche la configuration avant de faire l’édition	D	D
Change le nom de l’éditeur de liens cible	K, `gnatbl`	K, `gcc`
Change la taille maximale en caractères des lignes produites	l, `80`	l, `80`
Ajoute un chemin de recherche pour les fichiers objets des bibliothèques.	L	L
Change la valeur de `MAXINT`	M, `2147483647`	M, `2147483647`
Change la valeur de `MININT`	N, `-2147483647`	m, `-2147483647`
Demande une édition de liens pour projet hétérogène	n	n
Change le suffixe des fichiers objets de type “B Link File”	O, `blf`	O, `blf`
Change le répertoire où les fichiers sont générés	P, `../lang`	P, `../lang`
Change le suffixe des fichiers objets de type “spécifications de paquetage”	S, `str`	S, `spe`
Change la valeur en caractères d’une tabulation	t, `4`	t, `4`
Génère du code Ada dont l’exécution produit des traces sur les opérations appelées et les valeurs des paramètres	T	non dispo
Mode verbeux	v	v
Numéro de version et usage du logiciel	V	V

FIG. 5.3 – Options utilisables lors de l’édition des liens d’un projet

5.4 Restriction d'usage des traducteurs

Avertissement : Les traducteurs ADA, C et C++ fournis dans la version 3.6 de l'Atelier B sont expérimentaux. Leur but est de montrer qu'il est possible de traduire des implémentations B0 en langages de programmation classiques. Leur utilisation n'est donc pas garantie.

Le but de cette section est de définir l'ensemble des anomalies connues sur les traducteurs :

- Nom de paquetage :
  Les traducteurs ADA acceptent de générer des paquetages ayant le même nom que des paquetages prédéfinis du langage. Le compilateur Ada détecte donc un conflit et donc, ces paquetages ne peuvent donc pas être compilés. Pour contourner cette anomalie, il suffit de donner aux composants B un nom différent de ceux des paquetages prédéfinis ADA.
- Traduction des tableaux :
  La traduction en ADA des tableaux à l'aide des composants réutilisables (les machines de la famille L_ARRAY) peut être incorrecte. Le traducteur ADA peut allouer un espace très grand pour le tableau (integer) et donc saturer la mémoire.
- Comparaison de champs de records de type énuméré :
  Lorsqu'on compare ('=' ou '/=') 2 énumérés ou tableaux ou records, qui sont eux-memes des champs de record provenant d'une machine extérieure, les traducteurs traduisent : rec'champ = enum au lieu de : mch."="(rec'champ , enum).
- Paramètres formels ensembles :
  La traduction des paramètres formels ensembles par les traducteurs fournis avec l'Atelier B est parfois incorrecte. Il est déconseillé d'utiliser le même nom pour différents paramètres formels ensembles.
- Utilisation des composants réutilisables :
  Les composants réutilisables fournis avec l'Atelier B utilisent des paramètres formels ensemble. Conséquence de l'anomalie pré-citée, leur utilisation avec les traducteurs ADA, C et C++ peut aboutir sur une erreur du traducteur ou à une code cible erroné qui ne peut donc pas s'exécuter.

Chapitre 6

Glossaire

API Application Program Interface. Interface extérieure offerte par un système ou une bibliothèque.

Code offensif Un code offensif est un code où l'on suppose que l'utilisateur respecte un "contrat", i.e. que certaines conditions sont respectées. Ainsi, on ne teste pas ces conditions dans le code. En C, les fonctions du type `strcpy` et `strcmp` sont des exemples de code offensif car elles ne vérifient pas l'intégrité de leurs arguments *(zones mémoires allouées par l'utilisateur, chaines terminées par* `\0`*, ...)*.

IHM Interface Homme-Machine. Cette interface est graphique, l'interface en mode texte par ligne de commande est le mode batch.

Mode batch Mode de fonctionnement par ligne de commande de l'Atelier B. Ce mode permet d'exécuter des procédures automatiques décrites sous formes de fichiers de commandes.

SCCS *(Source Code Control System)*. Ensemble d'outils permettant de gérer la configuration de fichiers sources et, par extension, des fichiers binaires résultant de la compilation de ces fichiers sources.

Shell Programme d'interface entre l'utilisateur et le système d'exploitation. Sous Unix, les principaux shells sont `sh`, `ksh`, `bash` et `csh`.

Annexe A

Les machines de base

A.1 Principe

A.1.1 Définition

Une machine de base est une machine qui possède une spécification en B et qui est directement implantée dans le langage cible

A.1.2 Utilité

Les machines de base sont utilisées pour implanter des fonctionnalités qui ne peuvent pas être réalisées en B0. Le plus souvent, ces fonctionnalités sont des fonctionnalités proche du système : entrées/sorties, gestion dynamique de mémoire, ...

L'Atelier B est livré en standard avec une collection de machines de base qui permettent de réaliser des projets B qui interagissent avec un opérateur et qui utilisent des structures de données complexe. Le paragraphe A.2 décrit ces machines. Pour autant, un utilisateur de l'Atelier peut avoir besoin d'implémenter ses propres machines de base. Le paragraphe A.3 donne la marche à suivre pour réaliser cette tâche.

A.2 Description des machines de base livrées avec l'Atelier B

Le tableau A.1 donne la liste des machines de base livrées avec l'Atelier B. Le manuel [ATB2] donne une description complète de ces machines : le lecteur désireux de trouver plus d'informations sur leur utilisation pourra s'y reporter.

Les machines de base sont les seuls moyens offerts à l'utilisateur d'implanter certaines des fonctionnalités des spécifications. Ainsi, la seule façon d'implanter un tableau "dynamique" [1] est l'importation de `BASIC_ARRAY_VAR` (une dimension) ou `BASIC_ARRAY_RGE` (deux dimensions).

[1] i.e. un tableau dont la taille dépend des paramètres de la machine.

MACHINE	DESCRIPTION
`BASIC_ARRAY_RGE`	Implantation d'un tableau a deux dimensions
`BASIC_ARRAY_VAR`	Implantation d'un tableau a une dimension
`BASIC_IO`	Entrées/Sorties de base

FIG. A.1 – Machines de base livrées avec l'Atelier B

A.3 Méthode d'écriture d'une machine de base

Une machine de base écrite par l'utilisateur est composée de quatre éléments :

1. Une spécification en B,
2. Une interface (ou spécification), écrite en langage cible,
3. Un corps de paquetage (ou de classe), écrit en langage cible,
4. Un fichier au format "B Link File" destiné à l'éditeur de liens.

La méthode la plus simple pour écrire une machine de base est :

1. Ecrire la spécification en B,
2. Ecrire une implémentation coquille vide, c'est à dire où toutes le corps des opérations contient `skip`,
3. Faire traduire l'implémentation par l'atelier B,
4. Garder le fichier B link file et remplir les squelettes obtenus par le code souhaité.

A.3.1 Méthode d'écriture de la spécification B

La spécification B est écrite en suivant les règles usuelles de l'écriture d'un composant en B. Il est intéressant d'écrire une spécification qui décrit aussi précisément que possible l'effet des opérations de la machine de base, plutôt que de se limiter à donner des coquilles vides *(i.e.* `skip`*)* pour spécification. Ainsi, le mécanisme de preuve permet de garantir à l'auteur du code en langage cible de la machine de base qu'un certain nombre de contraintes son respectées, et donc d'écrire du code offensif* .

A.4 Méthode d'écriture du code cible (spécification et corps)

Le code cible doit être composé d'une spécification et d'un corps de paquetage, localisé dans deux fichiers. Les extensions respectives par défaut de ces deux fichiers sont données par la table 5.2. Si l'on utilise les options du Traducteur permettant de changer ces extensions[2], on veillera à répercuter ce changement sur les suffixes des fichiers composant les machines de base.

Les paragraphes A.4.1, A.4.2, A.4.3 et A.4.4 explicitent les méthodes d'écriture de code cible pour les traducteurs Ada, HIA, C++ et C respectivement.

[2]Ces options sont décrites au chapitre 5.

Type B0	Type Ada/HIA	Type C/C++
INT	INTEGER	T_int
NAT	INTEGER	T_nat
NAT1	INTEGER	T_nat
BOOL	BOOLEAN	T_bool
STRING	STRING	T_string
paramètre formel de type ensemble	↻ Elément de paquetage générique (ADA) ↻ Sous-type (HIA)	T_set *
tableau	array	T_array_x * x = dim. du tableau

FIG. A.2 – Traduction des types B0 en Ada, HIA, C et C++

A.4.1 Méthode d'écrire du code cible en Ada

↻ CETTE SECTION NE S'APPLIQUE QU'AU TRADUCTEUR ADA

– Le nom du paquetage est le même que le nom de la spécification B, en utilisant indifféremment des majuscules ou des minuscules. Par contre, les noms de fichiers utilisés doivent être en minuscules. Soit P ce nom.
– Le paquetage P doit définir :
  – Le type TYPE_P qui est un record qui doit comporter obligatoirement un champ initialisation de type BOOLEAN. Dans ce record, on place toutes les structures de données nécessaires à la modélisation des instances de chaque machine.
  – Le type PTR_P, de type "pointeur sur P"
  – Une procédure IMPORTS qui prend en entrée un paramètre this de type PTR_P et qui décrit l'instance à importer, puis autant de paramètres en entrée que de paramètres formels de type scalaire. La figure A.2 donne la correspondance entre les types B0 et les types en langage cible.
  – Une procédure INITIALISATION qui prend en entrée un paramètre this de type PTR_P et qui décrit l'instance à importer.
  – Autant de procédures que d'opérations de la machine. Les procédures ont le même nom que dans la spécification B, précédé du nom de la machine en minuscules, encadré par des caractères # *(i.e. selon le format* #machine#operation*)*. On obtient leurs paramètres de la façon suivante :
    – Le premier paramètre, dit d'instance implicite, de type TYPE_P qui pointe sur l'instance du composant sur laquelle on effectue l'opération.
    – Suivent ensuite les paramètres d'entrée de l'opération, dans l'ordre de leur déclaration en B. Ce sont des paramètres de mode in. Les types sont obtenus conformément au tableau de la figure A.2.
    – Suivent ensuite les paramètres de sortie de l'opération, dans l'ordre de leur déclaration en B. Ce sont des paramètres de mode in out. Les types sont obtenus conformément au tableau de la figure A.2.

A.4.2 Méthode d'écriture de code cible en HIA

↻ CETTE SECTION NE S'APPLIQUE QU'AU TRADUCTEUR HIA

– Le nom du paquetage est le même que le nom de la spécification B, en utilisant indifféremment des majuscules ou des minuscules. Par contre, les noms de fichiers utilisés doivent être en minuscules. Soit P ce nom.
– Le paquetage P doit définir :
  – Une clause `with` pour chaque machine requise.
  – La traduction de la procédure `INITIALISATION`, placée dans la fonction d'initilisation du paquetage.
  – Autant de procédures que d'opérations de la machine. Les procédures ont le même nom que dans la spécification B, précédé du nom de la machine en minuscules, encadré par des caractères `#` *(i.e. selon le format* `#machine#operation`*)*. On obtient leurs paramètres de la façon suivante :
    – Les paramètres d'entrée de l'opération, dans l'ordre de leur déclaration en B. Ce sont des paramètres de mode `in`. Les types sont obtenus conformément au tableau de la figure A.2.
    – Les paramètres de sortie de l'opération, dans l'ordre de leur déclaration en B. Ce sont des paramètres de mode `in out`. Les types sont obtenus conformément au tableau de la figure A.2.

A.4.3 Méthode d'écriture de code cible en C++

↻ Cette section ne s'applique qu'au traducteur C++

– Le nom de la classe est le nom de la spécification B prefixée par `T_`. Soit P ce nom.
– La classe T_P doit définir :
  – la classe T_P qui doit comporter obligatoirement un champ `initialisation` de type `T_bool`. Dans cette classe, on place toutes les structures de données nécessaires à la modélisation des instances de chaque machine.
  – Une fonction membre `IMPORTS` qui décrit l'instance à importer, puis autant de paramètres en entrée que de paramètres formels de type scalaire ou ensembliste. La figure A.2 donne la correspondance entre les types B0 et les types C++.
  – Une fonction membre `INITIALISATION` qui décrit l'initialisation de l'instance.
  – Autant de fonctions membre que d'opérations de la machine. Les procédures ont le même nom que dans la spécification B, précédé du nom de la machine en minuscules, encadré par des caractères `#` *(i.e. selon le format* `#machine#operation`*)*. On obtient leurs paramètres de la façon suivante :
    – Les paramètres d'entrée de l'opération, dans l'ordre de leur déclaration en B. Les types[3] sont obtenus conformément au tableau de la figure A.2.
    – Suivent ensuite les paramètres de sortie de l'opération, dans l'ordre de leur déclaration en B. Les types sont obtenus conformément au tableau de la figure A.2.

A.4.4 Méthode d'écriture de code cible en C

↻ Cette section ne s'applique qu'au traducteur C

– Le nom de la pseudo classe est le nom de la spécification B préfixée par `T_`. Soit P ce nom. *(C'est en fait une structure à laquelle on accède uniquement aux travers de fonctions, préfixées par le nom de la classe, qui émulent les méthodes d'une véritable classe).*

[3] Les classes T_set et T_array_X sont définies dans des composants prédéfinis fournis avec le traducteur.

– La pseudo classe T_P doit définir :
  – Un champ `initialisation` de type `T_bool`.
  – Un champ par données nécessaires à la modélisation de la machine (paramètre formel, variable concrète, ...). Les noms des champs sont préfixés par le nom de la machine entre caractères `#`.
– Les méthodes de la classe T_P sont émulées par les fonctions suivantes :
  – Une fonction `new_T_P`. C'est l'équivalent du constructeur de la classe. Elle prend en entrée un paramètre `_this` de type `T_P *`.
  – Une fonction `T_P_IMPORTS`. Elle prend en entrée un paramètre `_this` de type `T_P *`, puis autant de paramètres en entrée que de paramètres formels de type scalaire ou ensembliste. La figure A.2 donne la correspondance entre les types B0 et les types C.
  – Une fonction `T_P_INITIALISATION`. Elle prend en entrée un paramètre `_this` de type `T_P *`. Elle décrit l'initialisation de l'instance.
  – Autant de fonctions que d'opérations de la machine. Ces fonctions ont le nom que l'opération dans la spécification B, préfixé du nom de la pseudo classe et précédé du nom de la machine, lui-même encadré par des caractères `#` *(i.e. selon le format* `#machine#T_P_operation`*)*. On obtient leurs paramètres de la façon suivante :
    – Le premier paramètre est `_this` de type `T_P *`.
    – Les paramètres d'entrée de l'opération, dans l'ordre de leur déclaration en B. Les types[4] sont obtenus conformément au tableau de la figure A.2.
    – Suivent ensuite les paramètres de sortie de l'opération, dans l'ordre de leur déclaration en B. Les types sont obtenus conformément au tableau de la figure A.2.

A.4.5 Le fichier "B Link File"

Le fichier "B link File" est composé de plusieurs sections délimitées par les mots clés `_BEGIN_NOM_SECTION` et `_END_NOM_SECTION`. Ces sections peuvent être vides ou absentes. Elles contiennent des informations permettant de gérer les conflits de noms lors de l'édition de liens entre les différents modules ainsi que les conflits avec le langage cible. Elles permettent aussi de répertorier les types tableaux et les constantes exportées par le module.

Si on suppose que le code de la machine de base est un code correct, utilisant des identificateurs valides pour le langage cible choisi, il suffit d'indiquer le nom de la machine (section `CLASS`) et de remplir la section des noms des opérations exportées par la machine (sous section `LEVEL_1` de la section `CLASS`).

Ainsi les noms des opérations exportées par la machine de base n'entreront pas en conflit avec des noms d'autres modules.

☉ Attention, pour les traducteurs Ada et HIA, et contrairement aux conventions du langage Ada, le nom des opérations doit respecter l'utilisation des majuscules et des minuscules qui est faite dans le source B de la spécification de la machine de base.

La figure A.3 donne l'exemple du fichier "B Link File" fourni pour la machine de base `BASIC_IO`.

[4] Les classes T_set et T_array_X sont définies dans des composants prédéfinis fournis avec le traducteur.

```
--*****************************************************************************
--
--Base machine BASIC_IO for ADA target language
--
--(C) 2001 ClearSy
--
--Version @(#) basic_io.blf 1.2 (date : 22 Mar 1996)
--
--*****************************************************************************
_BEGIN_CLASS
   -- Machine name
   BASIC_IO
   _BEGIN_LEVEL_1 -- Class level
      -- operations
      INTERVAL_READ
      INT_WRITE
      BOOL_READ
      BOOL_WRITE
      CHAR_READ
      CHAR_WRITE
      STRING_WRITE
   _END_LEVEL_1
_END_CLASS
```

FIG. A.3 – Exemple de fichier "B Link File"'

Annexe B

Spécificités du language B0 accepté par le traducteur HIA

B.1 Introduction

Le traducteur HIA utilise un langage B0 qui lui est propre. Ces spécificités sont causées :

– D'une part, par le typage explicite par identificateur des tableaux et des articles,
– D'autre part, par la volonté de générer un code simple, et en particulier des paquetages Ada proches des composants B, regroupant notamment les déclarations des constantes et des paramètres formels

Nous désignerons par B0-HIA ce langage dans la suite de ce chapitre.

B.2 Traduction des tableaux

B.2.1 Principe

Pour pouvoir être traduite par le traducteur HIA, une donnée de type tableau doit impérativement avoir été typée explicitement par un identificateur, c'est à dire par appartenance à une donnée qui définit un type tableau *(c'est une restriction par rapport au langage B0 classique)*

Les données qui définissent les types tableaux sont des constantes concrètes, qui doivent être typées par égalité avec un type "tableau" B. C'est ce type B définit dans la clause `PROPERTIES` qui est utilisé pour la traduction ; la valuation est ignorée.

Nous préconisons de recopier le typage de la clause `PROPERTIES` dans la clause `VALUES` pour éviter toute confusion. La possibilité de définir une constante concrète de type tableau est une extension par rapport au langage B0 classique

B.2.2 Exemple

Pour écrire en B0-HIA la déclaration d'une variable var de type « tableau d'entiers, dont les index sont entre 4 et 12 », on écrira :

```
CONCRETE_CONSTANTS
```

```
 type_tableau

PROPERTIES
 type_tableau = (4..12) --> INT  / * extension B0-HIA * /

VALUES
 type_tableau = (4..12) --> INT / * inutilisé par le traducteur * /

CONCRETE_VARIABLES
 var

INVARIANT
 var : type_tableau / * typage explicite par identificateur imposé par B0-HIA * /

INITIALISATION
     var := (4..12)*{1}
```

B.3 Traduction des articles

B.3.1 Principe

Pour pouvoir être traduite par le traducteur HIA, une donnée de type article doit impérativement avoir été typée explicitement par un identificateur, c'est à dire par appartenance à une donnée qui définit un type article *(c'est une restriction par rapport au langage B0 classique)*.

Les données qui définissent les types articles sont des constantes concrètes, qui doivent être typées par égalité avec un type "article" » B. C'est ce type B définit dans la clause `PROPERTIES` qui est utilisé pour la traduction ; la valuation est ignorée.

Nous préconisons de recopier le typage de la clause `PROPERTIES` dans la clause `VALUES` pour éviter toute confusion. La possibilité de définir une constante concrète de type tableau est une extension par rapport au langage B0 classique.

B.3.2 Exemple

Pour écrire en B0-HIA la déclaration d'une variable var de type « article, dont le premier champ tab est de type 'tableau d'entiers, dont les index sont entre 4 et 12', et le deuxième champ valid est de type 'booléen' », on écrira :

```
CONCRETE_CONSTANTS
 type_tableau,
 type_article

PROPERTIES
 type_tableau = (4..12) --> INT  / * extension B0-HIA * / &
 type_article = struct(
  tab : type_tableau,
  valid : BOOL) / * extension B0-HIA * /

VALUES
 type_tableau = (4..12) --> INT ; / * inutilisé par le traducteur * /
 type_article = struct(
  tab : type_tableau,
```

```
  valid : BOOL) / * inutilisé par le traducteur * /

CONCRETE_VARIABLES
 var

INVARIANT
 var : type_article / * typage explicite par identificateur imposé par B0-HIA * /

INITIALISATION
     var := rec( (4..12)*{1}, FALSE)
```

B.4 Paramètres formels

Les paramètres formels effectifs sont traduits directement dans le paquetage de l'instance concernée. Par exemple, si `Mch` est un composant B possédant un paramètre formel scalaire `param` et que dans le projet on crée deux instances `i1.Mch(5)` et `i2.Mch(10)` de `mch`, alors :

– dans le paquetage `i1_Mch` on définit une constante `param` de valeur `5`
– dans le paquetage `i2_Mch` on définit une constante `param` de valeur `10`

Les paramètres formels effectifs sont donc déclarés dans les paquetages associés. Par conséquent, ils ont la portée de ces paquetages.

Supposons maintenant qu'un composant `Mch1` utilise un composant `Mch2(param2)`, et que la valuation de `param2` fait intervenir des données de `Mch1`, par exemple une constante `cst1` de `mch1` : `param2 = Mch1.cst1`

La valeur de `Mch2.param2` fait donc intervenir `Mch1.cst1`. Or `Mch1.cst1` n'est pas visible depuis le paquetage `Mch2` *(et ne peut pas l'être : comme* `Mch1` *utilise* `Mch2`*, alors le paquetage* `Mch1` *fait un* `with` *de* `Mch2` *:* `Mch2` *ne peut donc pas faire de* `with` *de* `Mch1` *sous peine de dépendance cyclique).*

Il n'est donc pas possible de compiler le code obtenu. Plus généralement, on voit qu'il n'est pas possible d'utiliser des données non littérales pour la valuation des paramètres formels de machine.

Conséquence : Les paramètres formels effectifs de machines autorisés en BO-HIA sont les littéraux (entiers ou booléens).

Bibliographie

[ANSI-C++] Le langage C++, troisième édition
Bjarne STROUSTRUP
Campus Press
ISBN 2-7740-0609-2

[ANSI-C] Le langage C, norme ANSI
B.W. KERNIGHAN et D.M. RITCHIE

[ADA-83] Manuel de référence du langage de programmation Ada
Alsys
Février 1987

[ADA-95] Ada 95
Deuxième édition
John BARNES
Addisson-Wesley
ISBN 0-201-34293-6

[ADA2] Ada, An Introduction. Ada reference manual.
Henry Ledgard
Springer-Verlag
ISBN 0-387-90568-5 et 3-540-90568-5

[ADA3] Ada programming manual
Integrated computer systems
NEW/8/80

[SPARK] High integrity Ada
The SPARK approach
John BARNES
Addisson-Wesley
ISBN 0-201-17517-7

[ATB1] Manuel de référence de l'Atelier B.

[ATB2] Composants réutilisables. Manuel de référence.

Index

édition des liens, 8, 9, 13, 31

autonome, 19

B0
    implémentation, 5
    vérificateur, 5
Batch
    mode de l'Atelier B, 12, 15
bibliothèque, 6

classe C++
    corps, 8
    spécifications, 8
compilation, 5, 6, 14
    fichier makefile du projet, 9
composant C
    corps, 8
    spécifications, 8
conflits d'identificateurs, 5, 6

environnement, 5
    HP-UX, 5
    Linux, 5
    Solaris, 5
    Sun Os, 5

fichiers objets, 6, 8

hétérogène, 26, 28
HP-UX, 5

IHM de l'Atelier B, 11, 15
implémentation B0, 5
instance implicite, 28, 39

Linux, 5

machine de base, 9, 37
    écrite par l'utilisateur, 38
    code offensif, 38
    livrée avec l'Atelier B, 37
makefile, 9
Mode Batch de l'Atelier B, 12, 15
module d'accès, 26
module de démarrage, 26
module de lancement, 8

paquetage Ada
    corps, 8
    spécifications, 8
paquetage d'accès, 26
point d'entrée, 9, 24, 26
preuve, 16
projet
    module d'accès, 26
    module de démarrage, 26
    module de lancement, 8
    paquetage d'accès, 26
    point d'entrée, 9, 24, 26

Restrictions sur les noms de modules et de projets, 17

sets, 9
Solaris, 5
Sun Os, 5

traduction, 5, 8, 11, 12, 31
    article, 7
    avec traces, 31
    collage implicite, 7
    compatibilité, 5
    croisée, 5
    et preuve, 16
    MAXINT, 16
    MININT, 16
    point d'entrée, 9, 26
    portabilité, 5
    renommage, 7
    tableau, 7
    taille des lignes, 16

vérificateur aux limites, 16
vérificateur de B0, 5